Formal Semantics for Program Paths

Karl Lermer[†] and Colin Fidge[†] and Ian Hayes[‡]

[†]*Software Verification Research Centre*
[‡]*School of Information Technology and Electrical Engineering*
The University of Queensland, Queensland 4072, Australia.

Abstract

This paper provides the syntax and semantics for a systematic approach to the problem of analysing control-flow paths in computer programs. We give an abstract syntax and a partial correctness semantics for program control-flow paths as a generic model for path analysis and constraint derivation. This approach is formally based on a predicate transformer semantics over a boolean-valued predicate space and an abstract command language. The notions of a command, dead commands, the entry and exit conditions of a command and the inverse of a command are formally defined and investigated on the base of the semantics. A notion of command refinement is introduced capturing the abstraction process in program development from specification to implementation with partial correctness. Furthermore, command-reduction theorems and characterisations for command refinement are derived using the underlying semantics. Finally we verify the equivalence of weakest liberal precondition and strongest postcondition semantics for program commands in terms of the ordering relation they define on the command language. The approach is generic in that it is applicable to any program language that can be supplied with a predicate transformer semantics.

Keywords: Control-flow path analysis; Partial correctness semantics; Path refinement; Weakest liberal precondition semantics; Strongest postconditions.

1 Introduction

A program path is a sequence of statements that may be performed during the execution of a program. Extracting such control-flow paths from program code, and then studying them in isolation, is a fundamental concept in static code analysis, program testing, and performance prediction.

Just as weakest preconditions can be used to give a formal semantics to program code, so too weakest liberal preconditions can be used to formalise the effect of following a particular control-flow path. In this paper we investigate this concept in depth, to provide a sound basis for the development of program analysis theories and algorithms. In particular, our own research is motivated

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

```
a)  if n ≥ 0 then
b)     n := 2 * n
c)  else
d)     n := −1
e)  endif;
f)  if n < 0 then
g)     n := n + 1
h)  else
i)     n := 0
j)  endif
```

Fig. 1. Example program fragment.

by the need to identify timing constraints imposed by the programmer on the executable code [18,16,11].

For instance we are interested in 'entry' and 'exit' conditions of control-flow paths. The entry condition of a program path characterises the set of initial states from which the path can terminate, and the exit condition characterises the set of final states that can be reached when the path is taken. We do not observe nontermination at all which motivates the use of a partial correctness semantics. Weakest liberal preconditions can be used to derive entry conditions and strongest postconditions can be used to derive exit conditions of control-flow paths.

As a concrete example consider the program fragment in Figure 1. Although trivial, even this simple example illustrates several important points. There are four control-flow paths through this program. For instance, one of these follows the first branch of the first 'if' statement and the second branch of the second. Two of the remaining paths have empty exit conditions, they can never be executed at run-time, in other words, they are 'dead paths'.

Here we develop a semantics of program paths from first principles. A minimal set of path constructors is introduced consisting of: specification statements, to model arbitrary blocks of sequential code; variable declarations, to allow the state space to be modified; nondeterministic choice for introducing repetition of subpaths; and sequential composition, as the basic constructor of paths from subpaths. A weakest liberal precondition semantics is then developed for this path language. The semantics incorporates a notion of "context" predicates to provide essential information, such as the types of variables, about the surrounding program within which the path resides.

An alternative semantics in terms of strongest postconditions is then devised, which allows paths to be analysed in reverse. Since the weakest liberal precondition semantics allows for the computation of path entrance conditions, the strongest postcondition semantics allows for the computation of path exit conditions. A notion of refinement (semantics preserving transfor-

mation) is introduced, based on the partial correctness semantics of program paths founded on weakest liberal preconditions and strongest postconditions. The equivalence of the two path semantics in terms of the ordering relations they define on the path language is then formally proven.

1.1 Related work

Our work builds on, and consolidates, previous experience with program analysis and semantics. Firstly, we observe that path analysis is a fundamental concept in writing and studying computer programs. It is the basis of static analysis techniques for identifying ineffective assignments and other coding errors [5], for ensuring adequate code coverage during testing [22], and for predicting program performance [6].

It has also long been recognised that program semantics can provide a formal basis for path analysis techniques and algorithms. For instance, the Path Exploration Tool (PET) uses symbolic execution techniques to calculate the precondition required to successfully execute a particular (manually chosen) control-flow path [14]. Symbolic execution is also used by a number of algorithms that aim to identify "dead" or "infeasible" paths, i.e., those that can never be followed at run time [1]. Many execution-time analysis techniques use a "semantics" of timing "facts" that can be derived from statements in each path [10].

Particularly relevant to our work are those techniques explicitly based on weakest liberal and weakest precondition semantics. For instance, the SPADE program analysis tool calculates the conditions required to successfully traverse a control-flow path using an algorithm whose correctness is justified in terms of weakest preconditions [5,6].

Weakest precondition semantics (total correctness) itself was succinctly introduced by Dijkstra [7]. He also described weakest liberal preconditions (partial correctness) in which termination is assumed, rather than required, and we use this as the starting point for our theory, on the expectation that only paths that have been correctly extracted from the program and shown to be feasible [15] will be studied. Strongest postconditions and program inverses (otherwise called program adjoints) have been around and studied for some time [12,8]. Dijkstra and Scholten [9] used strongest postcondition semantics and the notion of converse predicate transformers to prove a Galois connection between strongest postconditions and weakest liberal preconditions and to ultimately show the equivalence of these program semantics. Back and von Wright [4,23] defined and used program inverses and strongest postconditions to ultimately characterise program refinement in a weakest precondition (total correctness) approach. When analysing paths through compound constructs, such as 'if' or 'while' statements, we similarly adopt their use of 'coercions' as a way to model isolated parts of such constructs [3].

Finally, Morgan and Vickers [20] used predicates as program invariants in

a similar way to our use of predicates as contexts for program paths. The difference to our approach is that they work with a weakest precondition semantics.

1.2 Outline

Section 2 presents the boolean-valued predicate space that underlies the predicate transformer semantics for commands. In Section 3 we define the abstract syntax and weakest liberal precondition semantics for commands along with an abstract notion of command refinement. The latter captures the abstraction process during program development from a high-level program specification to implementation. The command language is based on four elementary command constructors, the specification statement, variable declarations, the nondeterministic command and sequential composition, together with an abstract construct to capture the context of a command. The underlying semantics is a partial correctness semantics over the boolean-valued predicate space. This language is expressive enough to model any constructs of a typical sequential imperative programming language. Section 4 looks at the command analysis problem from a different perspective. A strongest postcondition semantics for program commands is introduced allowing one to determine the program state after execution of a command. To express elementary correspondences between the weakest liberal precondition and strongest postcondition semantics the notion of the inverse of a path is introduced. The generality of the path language and semantics allows reduction of a program path to a specification statement. This is shown in Section 5. A Galois connection between weakest liberal precondition and strongest postcondition semantics for program commands in terms of command refinement is proven in Section 6 showing the equivalence of these two command semantics. Appendix A at the end of the paper contains important lemmas and theorems.

2 The predicate space

Let *Var* denote the universal set of variables that may take their values from a universal set of values *Val*. The set of all possible bindings *Bnd* is then defined as the set of all functions from *Var* to *Val*. We assume that for every unprimed variable $x \in Var$ there is also a primed variable identifier $x' \in Var$. We take the boolean space

$$\mathbb{B} \stackrel{def}{=} \{false, true\}$$

with the conventional ordering '$\trianglelefteq$', where $false \lhd true$. We assume $\mathbb{B} \subseteq Val$. The operators '$\wedge$', '$\vee$' and '$\neg$' are defined as the conventional boolean operators on the boolean algebra $\mathbb{B}$; '$\wedge$' denoting the infimum and '$\vee$' the supremum in $\mathbb{B}$. The set of predicates *Pred* is defined as the set of all total functions from *Bnd* to $\mathbb{B}$. With the pointwise extension of the ordering '$\trianglelefteq$' and the lattice operators '$\wedge$' and '$\vee$' from $\mathbb{B}$, *Pred* becomes a complete lattice. For two pred-

icates P_1 and P_2 the *entailment ordering* '$\Rrightarrow$' on the predicate lattice *Pred* is defined by

$$P_1 \Rrightarrow P_2 \text{ iff } P_1(\beta) \trianglelefteq P_2(\beta) \text{ for all } \beta \in \mathit{Bnd}$$

and $P_1 \equiv P_2$ iff $P_1 \Rrightarrow P_2$ and $P_2 \Rrightarrow P_1$. The additional predicate operators '$\Rightarrow$' and '$\Leftrightarrow$' are defined as $P \Rightarrow Q \stackrel{def}{=} Q \vee \neg P$ and $P \Leftrightarrow Q \stackrel{def}{=} (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

For a binding $\beta \in \mathit{Bnd}$, a subset of variables $\mathcal{V} \subseteq \mathit{Var}$ and a mapping $\sigma : \mathcal{V} \to \mathit{Val}$ we define the binding override $\beta[\sigma]$ by

$$\beta[\sigma](v) \stackrel{def}{=} \begin{cases} \beta(v) & \text{iff} \quad v \in \mathit{Var} \setminus \mathcal{V} \\ \sigma(v) & \text{iff} \quad v \in \mathcal{V} \ . \end{cases}$$

A function $f : \mathit{Bnd} \to \mathit{Val}$ may be *independent* of a variable v meaning that for every binding $\beta \in \mathit{Bnd}$ and value $r \in \mathit{Val}$ the equality $f(\beta) = f(\beta[v \mapsto r])$ holds, where $v \mapsto r$ denotes the mapping of variable v to value r. The set of all independent variables of f will be denoted by $\mathit{Notfree}(f)$. Variable v is called a *free variable* of f if $v \in \mathit{Var} \setminus \mathit{Notfree}(f)$.

The term algebra is defined as usual from the set of variables *Var* and the set of function symbols *Fun* which represent total functions of a certain arity over the value space *Val*. Substitution of a variable v by a term t in a predicate or function P on *Bnd* is defined by

$$P[t/v](\beta) \stackrel{def}{=} P(\beta[v \mapsto \beta(t)])$$

where the evaluation of a binding β on a term means the canonical extension of β onto the term algebra. By $P[y', z'/y, z]$ we denote the simultaneous substitution of y and z by y' and z', whereas $P[y'/y][z'/z]$ denotes the substitution of y by y' followed by the substitution of z by z'.

The operators '$\forall$' and '$\exists$' are defined on variables and predicates as follows, where $\beta \in \mathit{Bnd}$: $(\forall x\, P)(\beta)$ is defined as *true* if and only if $P(\beta[x \mapsto r]) = \mathit{true}$ for all $r \in \mathit{Val}$ and the existential operator '$\exists$' is defined by $\exists x P \stackrel{def}{=} \neg \forall x \neg P$. For any P, $P_i \in \mathit{Pred}$, $i \in \mathcal{A}$, where $\mathcal{A}$ is an arbitrary indexing set, the restricted distributivity law $(P \vee \wedge_{i \in \mathcal{A}} P_i) = \wedge_{i \in \mathcal{A}} (P \vee P_i)$ holds in the predicate space. Note that the extended de Morgan laws $\neg \wedge_{i \in \mathcal{A}} P_i \equiv \vee_{i \in \mathcal{A}} \neg P_i$ and the equality $\forall y \forall x\, P \equiv \forall x \forall y\, P$ hold for all predicates. The predicate $\forall y \forall x\, P$ shall be abbreviated by $\forall x, y\, P$. Note that for any term t where x does not occur free in t, the 'one-point rule' $\forall x (x = t \Rightarrow P) \equiv P[t/x]$ holds for all predicates $P \in \mathit{Pred}$.

Predicates in the semantic space *Pred* may have the entire set *Var* as free variables, because for example the set $\mathcal{A}$ in $\wedge_{i \in \mathcal{A}} P_i$ may be as large as *Var*. Such a predicate would not permit variable substitution with variables in the variable universe *Var* in the usual sense. To avoid pathological cases of this kind we need to make sure that predicates can not have more than a certain number of free variables. For a given cardinal number γ we therefore introduce the set Pred_γ which consists of all predicates P in *Pred* such that the

Table 1
Weakest liberal precondition semantics for commands.

Command S	$wlp(S, R)$
$(x\colon [Q])^C$	$\forall x'((C \wedge C[x'/x] \wedge Q) \Rightarrow R[x'/x])$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$(\forall\, v(wlp(S_1^{v\in T \wedge C[w/v]}, R[w/v])))[v/w]$ with $w \in \mathit{Notfree}(R)$ and $w \notin \mathit{Idf}(S)$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$\wedge_{i\in\mathcal{A}}\, wlp(S_i^C, R)$
$(S_1^{C_1})^{C_2}$	$wlp(S_1^{C_1\wedge C_2}, R)$
$(S_1\ ;\ S_2)^C$	$wlp(S_1^C, wlp(S_2^C, R))$
Command S	Free identifiers $\mathit{Idf}(S)$
$(x\colon [Q])^C$	all free variables in Q, C and the variables x, x'
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$\mathit{Idf}(S_1^{v\in T\wedge C}) \cup \{v, v'\}$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$\cup_{i\in\mathcal{A}} \mathit{Idf}(S_i^C)$
$(S_1^{C_1})^{C_2}$	$\mathit{Idf}(S_1^{C_1\wedge C_2})$
$(S_1\ ;\ S_2)^C$	$\mathit{Idf}(S_1^C) \cup \mathit{Idf}(S_2^C)$

cardinality of $\mathit{Var} \setminus \mathit{Notfree}(P)$ is not greater than γ. If γ is a limit ordinal, then Pred_γ is closed under infimum and supremum on sets of predicates with cardinality not greater than γ, i.e., for predicates P_i in Pred_γ, with index i ranging over an index set $\mathcal{A}$ of cardinality not greater than γ, it is the case that $\wedge_{i\in\mathcal{A}} P_i$ and $\vee_{i\in\mathcal{A}} P_i$ are predicates in Pred_γ. The semantics for program commands will be defined with predicates in Pred_γ for some cardinal γ. Wherever nondeterministic choice appears in the program language we additionally assume that γ is a limit ordinal.

A *predicate transformer* is defined as a function on predicates that is monotone with respect to the entailment ordering '$\Rrightarrow$'.

3 The command language

This section presents the syntax and semantics for a general language for program control-flow paths on the base of a partial correctness semantics. We are ultimately interested in program constraints like the program's timing behaviour or final state in case of program termination, but not in proving termination of the path itself. This justifies the use of a weakest liberal precondition semantics for the command language. We introduce a notion of path transformation (refinement) that ensures partial correctness during program development.

3.1 Syntax and semantics of a command

By a program path we mean any sequence S of statements that represents a possible execution of a given (high or low-level) program. Our command language therefore consists of elementary program constructs such as the specification statement, $x{:}\,[Q]$; a statement for the declaration of typed local variables, $(\mathbf{var}\, x\ {:}\, T \bullet S)$; nondeterministic choice over some set of program statements, $(\sqcap_{i\in\mathcal{A}} S_i)$; and sequential composition of program statements, $(S_1\ ;\ S_2)$. These constructs are sufficient to model control-flow paths through typical imperative programs. The specification statement can describe the behaviour of assignment and other primitive programming statements [19]. The variable declaration statement supports changes to the state space. Nondeterministic choice over program statements can be used to model conditional and iterative behaviour. Finally, sequential composition is the fundamental notion for constructing commands from subcommands. The predicates are from the predicate space $Pred_\gamma$ with some cardinal number γ which is a limit ordinal. The index set $\mathcal{A}$ for the nondeterministic choice statement is of cardinality not greater than γ.

We are interested in a semantics that allows us to determine the entry and exit condition of a program command. The entry condition describes the initial states from which termination is possible, and the exit condition describes the final states that can be reached in case of termination. We are not interested in termination issues commonly handled with weakest preconditions (total correctness). The case that the command will not be taken by the program is a separate concern [15]. Therefore we provide the language with a weakest liberal precondition semantics [9] (partial correctness). The standard semantics of a basic specification statement is as follows.

$$wlp(x{:}\,[Q], R) \stackrel{def}{=} \forall x'\,(Q \Rightarrow R[x'/x])$$

Such a specification statement updates the variables in its frame x according to the predicate Q. Note that the frame x stands for a (possibly empty) set of variables that may be changed by this statement. The primed list of variables x' denotes their final values. If the frame is empty we abbreviate specification $:[Q]$ as a *coercion* $[Q]$. Such statements can be used to document properties required to be true at a particular point in a path, especially the values of boolean expressions used as guards in conditional or iterative statements [15].

For our purposes we extend such conventional semantics with a 'context' C which is used to record information concerning the surrounding program from which a given command is extracted. Table 1 states the syntax and weakest liberal precondition semantics for the specification statement, the typed local variable declaration, the nondeterministic choice and the sequential composition of commands within a context. A *command* in our language then means any compound statement constructed from the commands in Table 1. In addition this table contains the definition of the set of variable identifiers free in

a command. Q, C and R always denote predicates in $Pred_\gamma$, where C and R are not permitted to have free primed variables. The predicate Q may have free unprimed variables and only the free primed variables x' corresponding to those that occur in the frame of the specification statement. In the interpretation, unprimed variables always refer to the state before, and primed variables to the state of the variable after, execution of the command. The type T of a variable can be any subset of the universal set of values *Val*.

Context predicate C is used to maintain invariant properties of the surrounding program throughout a command. Such invariants are typically type declarations for variables of the form $v \in T$. A command S is always seen in a certain (possibly $true$) context C expressed as S^C. For a specification statement the context is assumed to hold in the initial state and in the final state. Contexts in this sense are similar to the invariants introduced by Morgan and Vickers [20]. A variable declaration extends the context C by introducing the facts that new variable v has type T, and that the original context C holds in the presence of this newly-declared variable. A substitution with a fresh variable w is performed in context C and predicate R to exclude unwanted bindings to any externally declared variable with name v. Contexts are carried through nondeterministic choice, nesting and sequential composition in obvious ways.

Example 3.1 As a concrete example consider the program fragment in Figure 1. This simple example illustrates several important points. Firstly, the program variable n is global to this code fragment. Therefore information concerning n's declaration must form part of the context. In this case we assume n was declared as an integer and thus use context '$n \in \mathbb{Z}$'. Secondly, syntactic analysis reveals that there are four potential paths through this code. For instance, one of these follows the first branch of the first 'if' statement and the second branch of the second.

Path A:

a) $[n \geq 0]$
b) $n: [n' = 2 * n]$
f′) $[n \geq 0]$
i) $n: [n' = 0]$

The path is represented by a list of sequentially-composed specification statements. The first of these is a guard representing evaluation of the first 'if' statement's guard to 'true'. In this way we can decompose compound programming language statements into their primitives [15,3]. The second specification statement in Path A models assignment statement (b) in Figure 1. Recall that primed variables in postconditions denote final values. By translating programming language statements into specification constructs we minimise the number of constructs needed in the command language. Iterative statements appearing in commands, such as 'while' or 'for' loops, can be similarly modelled using the nondeterministic choice construct and iterated sequential

composition. Also notice that the third statement in Path A represents the guard on line (f) in Figure 1 evaluating to 'false', so the condition is negated. There are three other paths through this program (two of which are given below as paths B and C). The definitions and abbreviations in Table 1 allow us to calculate the semantics of typical program paths. Consider Path B below.

Path B:

$$\begin{aligned} &\text{a}') \quad [n < 0] \\ &\text{d}) \quad n\colon [n' = -1] \\ &\text{f}) \quad [n < 0] \\ &\text{g}) \quad n\colon [n' = n + 1] \end{aligned}$$

We can calculate its weakest liberal precondition with respect to a post-condition R, in context '$n \in \mathbb{Z}$', using the definitions in Table 1. For brevity, we firstly note that the semantics of a coercion construct with predicate Q and context C is as follows.

$$wlp([Q]^C, R) \equiv (C \wedge Q) \Rightarrow R$$

We then calculate the semantics of Path B by working backwards up the sequence of statements.

$$\begin{aligned} R_0 &\stackrel{def}{=} wlp((n\colon [n' = n + 1])^{n \in \mathbb{Z}}, R) \\ &\equiv \forall n' \left((n, n' \in \mathbb{Z} \wedge n' = n + 1) \Rightarrow R[n'/n]\right) \\ R_1 &\stackrel{def}{=} wlp(([n < 0])^{n \in \mathbb{Z}}, R_0) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow (n \in \mathbb{Z} \Rightarrow R[n + 1/n]) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow R[n + 1/n] \\ R_2 &\stackrel{def}{=} wlp((n\colon [n' = -1])^{n \in \mathbb{Z}}, R_1) \\ &\equiv (\forall n' \left((n, n' \in \mathbb{Z} \wedge n' = -1) \Rightarrow ((n \in \mathbb{Z} \wedge n < 0) \Rightarrow \right. \\ &\qquad \left. R[n + 1/n])[n'/n]\right)) \\ &\equiv n \in \mathbb{Z} \Rightarrow ((n' \in \mathbb{Z} \wedge n' < 0) \Rightarrow R[n' + 1/n])[-1/n'] \\ &\equiv n \in \mathbb{Z} \Rightarrow R[0/n] \end{aligned}$$

As the last step, we get the following predicate for the whole of Path B.

$$\begin{aligned} wlp((\text{Path B})^{n \in \mathbb{Z}}, R) &\equiv wlp(([n < 0])^{n \in \mathbb{Z}}, R_2) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow (n \in \mathbb{Z} \Rightarrow R[0/n]) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow R[0/n] \end{aligned}$$

In other words, Path B can achieve predicate R, in the given context, provided that the initial value of variable n is a negative integer and R holds with 0 substituted for n.

It is also important to recognise that some control-flow paths through a program are *dead* or *infeasible* because they can never be followed at run-time [15]. The *entry condition* of a command S is defined as the predicate $\neg wlp(S, false)$ which defines all states from where the program command will

be able to reach a final state. A command S is called *dead* if $wlp(S, false) \equiv true$. A command being dead means that its entry condition is 'false', or in other words that there are no states from which the command can reach a final state.

Example 3.2 For instance, Path C below is dead.

Path C:

a′) $[n < 0]$
d) $n\colon [n' = -1]$
f′) $[n \geq 0]$
i) $n\colon [n' = 0]$

Although syntactically valid Path C can never be followed at run-time, regardless of the initial value of variable n. We can prove this with the following calculation, again working backwards up the command.

$$\begin{aligned}
R_0 &\stackrel{def}{=} wlp((n\colon [n' = 0])^{n\in\mathbb{Z}}, false) \\
&\equiv (\forall n' ((n, n' \in \mathbb{Z} \wedge n' = 0) \Rightarrow false)) \\
&\equiv (\forall n' (n \notin \mathbb{Z} \vee n' \notin \mathbb{Z} \vee n' \neq 0)) \\
&\equiv n \notin \mathbb{Z} \\
R_1 &\stackrel{def}{=} wlp(([n \geq 0])^{n\in\mathbb{Z}}, R_0) \\
&\equiv (n \in \mathbb{Z} \wedge n \geq 0) \Rightarrow n \notin \mathbb{Z} \\
&\equiv n \notin \mathbb{Z} \vee n < 0 \\
R_2 &\stackrel{def}{=} wlp((n\colon [n' = -1])^{n\in\mathbb{Z}}, R_1) \\
&\equiv (\forall n' ((n, n' \in \mathbb{Z} \wedge n' = -1) \Rightarrow (n' \notin \mathbb{Z} \vee n' < 0)) \\
&\equiv n \in \mathbb{Z} \Rightarrow true \\
&\equiv true \\
R_3 &\stackrel{def}{=} wlp(([n < 0])^{n\in\mathbb{Z}}, R_2) \\
&\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow true \\
&\equiv true
\end{aligned}$$

Therefore we have $wlp((\text{Path C})^{n\in\mathbb{Z}}, false) \equiv true$ and the path is dead. The reason why the path can not be followed is revealed in the calculation for predicate R_2. At this point in the path the assignment $n := -1$ is followed by the impossible condition $n \geq 0$.

3.2 Liberal command refinement

A refinement relation on commands can be defined via the weakest liberal precondition semantics by interpreting a command as a predicate transformer on the subspace consisting of all predicates in $Pred_\gamma$ that do not have free primed variables. To maintain the distinct role of primed variables in this calculus it is necessary to assume that the predicates on which the predicate transformers are defined consist of unprimed variables only. All functions on

predicates that have been used to define the semantics from commands in Table 1 are monotone functions and the restrictions on the program statements ensure that the underlying predicate transformers map predicates from $Pred_\gamma$ without free primed variables to predicates in $Pred_\gamma$ without free primed variables.

Definition 3.3 [Liberal refinement] Let S_1, S_2 be two commands. We say that S_2 *liberally refines* S_1 and write $S_1 \sqsubseteq_{wlp} S_2$ if $wlp(S_1, R) \Rrightarrow wlp(S_2, R)$ for all predicates $R \in Pred_\gamma$ without free primed variables. Liberal refinement equivalence of S_1 and S_2 is denoted by $S_1 \sqsupseteq\sqsubseteq_{wlp} S_2$.

Note, that liberal refinement is different to common refinement relations based on weakest preconditions used in total correctness approaches [2,19,21]. A refinement $S \sqsubseteq_{wlp} S'$ in this weakest liberal precondition semantics means that the predicate characterising the states where command S' can start, i.e., $\neg wlp(S', \mathit{false})$, or in other words the 'entry condition' of command S', is stronger than the one for command S, and furthermore, that all states that command S' can achieve are possible ones for command S. Every liberal refinement decreases nondeterminism, strengthens the entry condition and/or increases the domain of nontermination (partial correctness).

Program development based on weakest preconditions increases the initial states from where the program is guaranteed to terminate and decreases the program's nondeterminism on the states from where termination is guaranteed (total correctness). Such a program development can increase the entry condition.

Example 3.4 To illustrate a path refinement we introduce the following short path. (This is not a path of the program in Figure 1.)

Path D:

$$\begin{array}{ll} x) & [n < 0] \\ y) & n\colon [n' = 0] \end{array}$$

We can then show that this path is a refinement of Path B in the context $n \in \mathbb{Z}$, i.e., $(\text{Path B})^{n\in\mathbb{Z}} \sqsubseteq_{wlp} (\text{Path D})^{n\in\mathbb{Z}}$ by calculating the semantics of Path D and comparing it to the semantics calculated for Path B above.

$$\begin{aligned} R_0 &\stackrel{def}{=} wlp((n\colon [n' = 0])^{n\in\mathbb{Z}}, R) \\ &\equiv \forall n' \left((n, n' \in \mathbb{Z} \wedge n' = 0) \Rightarrow R[n'/n]\right) \\ &\equiv n \in \mathbb{Z} \Rightarrow R[0/n] \end{aligned}$$

Therefore,

$$\begin{aligned} wlp((\text{Path D})^{n\in\mathbb{Z}}, R) &\equiv wlp(([n < 0])^{n\in\mathbb{Z}}, R_0) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow (n \in \mathbb{Z} \Rightarrow R[0/n]) \\ &\equiv (n \in \mathbb{Z} \wedge n < 0) \Rightarrow R[0/n] \\ &\equiv wlp((Path\, B)^{n\in\mathbb{Z}}, R) \end{aligned}$$

and we can conclude that not only is Path D a refinement of Path B in this

Table 2
The inverse of a command.

Command S	Inverse ${}^{o}S$
$(x\colon [Q])^C$	$(x\colon [Q[x, x'/x', x]])^C$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$(\mathbf{var}\, v\ : T \bullet {}^{o}S_1)^C$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$(\sqcap_{i\in\mathcal{A}} {}^{o}S_i)^C$
$(S_1^{C_1})^{C_2}$	${}^{o}S_1{}^{C_1\wedge C_2}$
$(S_1\ ;\ S_2)^C$	$({}^{o}S_2\ ;\ {}^{o}S_1)^C$

context, but they are refinement equivalent. Even further, a clever programmer would also recognise that the following path

Path E:

u) $[n \geq 0]$

v) $n\colon [n' = 0]$

is liberal refinement equivalent to Path A. Thus the whole program in Figure 1 could be optimised to the single-line program $n := 0$.

4 Inverses and strongest postconditions for commands

Above we defined a weakest liberal precondition semantics for commands. In this section we define command inverses and develop an alternative strongest postcondition semantics. Inverses, adjoints, duals and strongest postconditions for programs have been around for a long time in formal program reasoning [12,8,9,4,23,24]. Back and von Wright [4] used program inverses to characterise refinement based on weakest preconditions (total correctness) in terms of strongest postconditions. Dijkstra and Scholten [9] used strongest postconditions and converse predicate transformers to verify a Galois connection between weakest liberal preconditions and strongest postcondition (partial correctness).

We use inverses in a partial correctness approach to refinement to verify and express the equivalence of weakest liberal precondition and strongest postcondition semantics for program paths.

The inverse ${}^{o}S$ of a command S is defined by reversing all underlying relations and going from back to front in the command as depicted in Table 2. The inverses of all the constructors are straightforward except the (atomic) specification statement. The inverse of a specification that makes Q 'true' is defined here to be a specification that makes Q 'true' but with the roles of the initial and final variables reversed.

Example 4.1 As a simple example for the inverse of a path we state the inverse of Path B below.

Inverse of Path B:

g) $n\colon [n = n' + 1]$
f) $[n < 0]$
d) $n\colon [n = -1]$
a′) $[n < 0]$

For a given command S_2 it may be the case that a predecessor S_1 is known to always be executed before. This may influence the behaviour of command S_2 and thus it is necessary to develop a formal approach to incorporate this additional knowledge into path analysis. The assumptions that can be made about a predecessor are in general not invariant for the succeeding command and thus have to be incorporated in some other way than by using a command context. For this reason we define a strongest postcondition semantics for all command constructors of our path language.

The strongest postcondition of a specification statement $x\colon [Q]$ under the assumption R, where R is a predicate with no free primed variables, is defined by

$$sp(x\colon [Q], R) \stackrel{def}{=} (\exists x\,(R \wedge Q))[x/x'] \ .$$

Table 3 then defines the strongest postcondition semantics of a command S in a context C. The assumptions on the expressions and predicates are as before. In this situation, predicate R denotes the assumed precondition. For a specification statement this predicate is assumed to have held initially, along with initial and final versions of the context C, and the condition Q. The primed final-state variables are renamed as unprimed ones in order to form a 'single' state predicate. Existential quantification is used to hide the initial-state variables. The strongest postcondition of a variable declaration evaluates statement S_1 in a context where variable v is known to be of type T. The strongest postcondition of nondeterministic choice over a set of commands is defined as the disjunction of the strongest postconditions of the individual commands. The strongest postconditions of the remaining command constructors follow in an obvious way.

Example 4.2 We can calculate the strongest postcondition of Path B, in context '$n \in \mathbb{Z}$', with respect to a precondition R, by working forwards through the statements in the path.

$$\begin{aligned}
R_0 &\stackrel{def}{=} sp(([n < 0])^{n\in\mathbb{Z}}, R) \\
&\equiv n \in \mathbb{Z} \wedge n < 0 \wedge R \\
R_1 &\stackrel{def}{=} sp((n\colon [n' = -1])^{n\in\mathbb{Z}}, R_0) \\
&\equiv (\exists n(n \in \mathbb{Z} \wedge n' = -1 \wedge (n \in \mathbb{Z} \wedge n < 0 \wedge R)))[n/n'] \wedge n \in \mathbb{Z} \\
&\equiv (\exists n(n \in \mathbb{Z} \wedge n < 0 \wedge n' = -1 \wedge R))[n/n'] \wedge n \in \mathbb{Z} \\
&\equiv (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n])) \wedge n = -1 \\
R_2 &\stackrel{def}{=} sp(([n < 0])^{n\in\mathbb{Z}}, R_1)
\end{aligned}$$

Table 3
Strongest postcondition semantics of a command.

Command S	sp-Semantics $sp(S, R)$
$(x\colon [Q])^C$	$(\exists x\,(R \wedge C \wedge Q))[x/x'] \wedge C$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$(\exists v(sp(S_1^{v\in T \wedge C[w/v]}, R[w/v])))[v/w]$ with $w \in \mathit{Notfree}(R)$ and $w \notin \mathit{Idf}(S)$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$\vee_{i\in\mathcal{A}}\, sp(S_i^C, R)$
$(S_1^{C_1})^{C_2}$	$sp(S_1^{C_1\wedge C_2}, R)$
$(S_1\ ;\ S_2)^C$	$sp(S_2^C, sp(S_1^C, R))$

$$\equiv\ n \in \mathbb{Z} \wedge n < 0 \wedge (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n])) \wedge n = -1$$
$$\equiv\ n = -1 \wedge (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n]))$$

This gives us the following predicate for Path B.

$$\begin{aligned} sp((\text{Path B})^{n\in\mathbb{Z}}, R) &\equiv sp((n\colon [n' = n+1])^{n\in\mathbb{Z}}, R_2) \\ &\equiv (\exists n(n, n' \in \mathbb{Z} \wedge n' = n+1 \wedge n = -1 \wedge \\ &\qquad (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n]))))[n/n'] \\ &\equiv (n' \in \mathbb{Z} \wedge n' = 0 \wedge \\ &\qquad (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n])))[n/n'] \\ &\equiv n = 0 \wedge (\exists n''(n'' \in \mathbb{Z} \wedge n'' < 0 \wedge R[n''/n])) \end{aligned}$$

In other words, the strongest postcondition derivable from Path B, with respect to some precondition R, is that the final value of variable n is zero and there existed an initial value n'' which was a negative integer and that predicate R held with n'' substituted for n.

For a command S we call the predicate $sp(S, \mathit{true})$ the *exit condition* of S. This predicate describes all final states that can be reached by execution of the command. For a prefix S_1 of a command $(S_1\ ;\ S_2)$ the 'starting' condition for command S_2 is then computed by evaluating $sp(S_1, \mathit{true})$. More generally, if predicate R is known to hold at the start of S_1, $sp(S_1, R)$ describes all exit states of command S_1 under this assumption.

We can now define a notion of command refinement in terms of the strongest postcondition semantics. We show later that this notion is equivalent to liberal refinement as defined above.

Definition 4.3 [Sp-refinement] Let S_1, S_2 be two commands. We say that S_2 *sp-refines* S_1 and write $S_1 \sqsubseteq_{sp} S_2$ if $sp(S_2, R) \Rrightarrow sp(S_1, R)$ for all predicates $R \in \mathit{Pred}_\gamma$ without free primed variables. Sp-refinement equivalence of S_1 and S_2 is denoted by $S_1 \sqsupseteq\sqsubseteq_{sp} S_2$.

Example 4.4 Earlier we stated that Path B is refinement equivalent to Path D in the weakest liberal precondition semantics. Here we will undertake the

same proof in the strongest postcondition semantics, i.e., $(\text{Path B})^{n\in\mathbb{Z}} \sqsupseteq_{sp} (\text{Path D})^{n\in\mathbb{Z}}$. We calculated the strongest postcondition semantics for Path B above, so we now need to do the same for Path D.

$$\begin{aligned} R_0 &\equiv sp(([n<0])^{n\in\mathbb{Z}}, R) \\ &\equiv n\in\mathbb{Z}\wedge n<0\wedge R \\ sp((\text{Path D})^{n\in\mathbb{Z}}, R) &\equiv sp((n\colon[n'=0])^{n\in\mathbb{Z}}, R_0) \\ &\equiv (\exists n(n,n'\in\mathbb{Z}\wedge n'=0\wedge n<0\wedge R))[n/n'] \\ &\equiv \exists n''(n'', n\in\mathbb{Z}\wedge n=0\wedge n''<0\wedge R[n''/n]) \\ &\equiv n=0\wedge(\exists n''(n''\in\mathbb{Z}\wedge n''<0\wedge R[n''/n])) \\ &\equiv sp((\text{Path B})^{n\in\mathbb{Z}}, R) \end{aligned}$$

In fact, this is once again a refinement equivalence and thus conforms precisely with our findings in the weakest precondition semantics.

5 Reduction theorems and command refinement

The generality of our constructs for commands leads to elegant reduction theorems. In this section we show that commands are conjunctive and that any command in our language can be reduced to a semantically equivalent specification statement and we verify an elementary characterisation of command refinement.

Theorem 5.1 (Command reduction) *Let S be a command. Then there is a specification statement S' such that $S \sqsupseteq_{wlp} S'$ holds.*

Proof. This follows from the definition of commands in Table 1 and Lemmas A.2, A.4, A.5 and A.6 in the Appendix. □

Next we verify that all commands are conjunctive and invariant with respect to their contexts.

Theorem 5.2 (Conjunctivity) *Let S be a command, let C be a context predicate and let R_i, $i\in\mathcal{A}$ be predicates without free primed variables. Then, the following assertions hold:*

i) $wlp(S^C, R) \equiv wlp(S^C, R\wedge C) \equiv C \Rightarrow wlp(S^C, R)$;

ii) $wlp(S, true) \equiv true$; *and*

iii) the function $\lambda R \bullet wlp(S,R)$ *is* $\wedge$*-continuous on predicates without free primed variables, i.e.,* $wlp(S, \wedge_{i\in\mathcal{A}} R_i) \equiv \wedge_{i\in\mathcal{A}} wlp(S, R_i)$.

Proof. Because of Theorem 5.1 it is sufficient to prove the assertions for a specification statement S of the form $x\colon[Q]$.

Assertion (i): For a predicate R without free primed variables and a context predicate C we have the following equivalences.

$$\begin{aligned} C \Rightarrow wlp(S^C, R) &\equiv C \Rightarrow \forall x'((C\wedge Q\wedge C[x'/x]) \Rightarrow R[x'/x]) \\ &\equiv \forall x'((C\wedge Q\wedge C[x'/x]) \Rightarrow R[x'/x]) \end{aligned}$$

$$\begin{aligned}
&\equiv wlp(S^C, R)\\
&\equiv \forall x'((C \wedge Q \wedge C[x'/x]) \Rightarrow (R \wedge C)[x'/x])\\
&\equiv wlp(S^C, R \wedge C)
\end{aligned}$$

Assertion (ii) is trivial and assertion (iii) proceeds as follows.

$$\begin{aligned}
wlp(S, \wedge_{i\in\mathcal{A}} R_i) &\equiv \forall x' \, (Q \Rightarrow \wedge_{i\in\mathcal{A}} R_i[x'/x])\\
&\equiv \forall x' \, (\neg Q \vee \wedge_{i\in\mathcal{A}} R_i[x'/x])\\
&\equiv \text{“Restricted Distributivity Law”}\\
&\quad \forall x' \ \wedge_{i\in\mathcal{A}} (\neg Q \vee R_i[x'/x])\\
&\equiv \wedge_{i\in\mathcal{A}} \forall x' \, (\neg Q \vee R_i[x'/x])\\
&\equiv \wedge_{i\in\mathcal{A}} wlp(S, R_i)
\end{aligned}$$

□

Having reduction theorems for the weakest liberal precondition and the strongest postcondition semantics in hand it is now straightforward to verify characterisations for command refinement in both semantics. Below we present a characterisation of liberal refinement.

Theorem 5.3 (Command refinement) *Let* $S \stackrel{def}{=} x\colon [Q]$ *and* $S' \stackrel{def}{=} x\colon [W]$ *be two commands. Then* $S \sqsubseteq_{wlp} S'$ *is equivalent to* $W \Rrightarrow Q$.

Proof. To show the equivalence between the refinement $S \sqsubseteq_{wlp} S'$ and the implication $W \Rrightarrow Q$ we separately consider both directions. First we assume $S \sqsubseteq_{wlp} S'$. With Lemma A.3, the refinement $S \sqsubseteq_{wlp} S'$ can be expressed by the following implication

(1) $\exists x' \, (W \wedge R[x'/x]) \Rrightarrow \exists x' \, (Q \wedge R[x'/x])$

for all predicates R without free primed variables. Let $b \in Bnd$ be a binding such that $true = W(b)$ holds. By defining the constant $r \stackrel{def}{=} b(x')$ and $R_0 \stackrel{def}{=} x = r$ the following holds for this predicate.

$$\begin{aligned}
true &= (\exists x' \, (W \wedge R_0[x'/x]))(b)\\
&= \text{ “By assumption (1)”}\\
&\quad (\exists x' \, (Q \wedge R_0[x'/x]))(b)\\
&= (\exists x' \, (Q \wedge x' = r))(b)\\
&= Q[r/x'](b)\\
&= Q(b)
\end{aligned}$$

This is all that we have to prove for this direction. To prove the equivalence in the reverse direction we assume that $W \Rrightarrow Q$ holds. But this implies that condition (1) holds for all predicates R without free primed variables and is all that we need to show for this direction. □

6 Correlations between semantics

The following theorem reveals the duality between a command and its inverse (assertions (i) and (ii)) and the so-called Galois connection between weakest liberal precondition and strongest postcondition semantics of a command (assertions (v) and (vi)). The latter has been proven by Dijkstra and Scholten with the help of so-called converse predicate transformers [9, Chapter 12, Theorem 2], for their abstract concept of weakest liberal and strongest postconditions of commands.

Theorem 6.1 (Command inverse duality) *Let S, S' be commands and R be a predicate without free primed variables. Then the following relationships hold.*

i) ${}^{o}{}^{o}S = S$

ii) $wlp(S, R) \equiv \neg sp({}^{o}S, \neg R)$

iii) $S \sqsupseteq_{wlp} [\neg wlp(S, false)]\ ;\ S$

iv) $S \sqsupseteq_{wlp} S\ ;\ [sp(S, true)]$

v) $R \Rrightarrow wlp(S, sp(S, R))$

vi) $sp(S, wlp(S, R)) \Rrightarrow R$

vii) $S \sqsubseteq_{sp} S'$ *is equivalent to* $S \sqsubseteq_{wlp} S'$

viii) $S \sqsubseteq_{wlp} S'$ *is equivalent to* ${}^{o}S \sqsubseteq_{wlp} {}^{o}S'$

Proof. Assertion (i): This elementary equality follows from the definition of substitution in predicates and the inverse of a command.

Assertion (ii): We show this condition by structural induction over the command language. For a specification statement S of the form $x\colon[Q]$ and a context C we obtain the inverse ${}^{o}S^{C}$ as $x\colon[Q[x, x'/x', x]]^{C}$ and so we have the following equivalence for any predicate R without primed variables.

$$\begin{aligned}
\neg sp({}^{o}S^{C}, \neg R) &\equiv \neg sp(x\colon[Q[x', x/x, x']]^{C}, \neg R) \\
&\equiv \neg(\exists x((Q \wedge C)[x, x'/x', x] \wedge C \wedge \neg R))[x/x'] \\
&\equiv \neg\exists x'(Q \wedge C \wedge (\neg R \wedge C)[x'/x]) \\
&\equiv \forall x'((Q \wedge C \wedge C[x'/x]) \Rightarrow R[x'/x]) \\
&\equiv wlp(S^{C}, R)
\end{aligned}$$

For two commands S_1 and S_2 the following equivalence for sequential composition holds.

$$\begin{aligned}
wlp((S_1\ ;\ S_2)^{C}, R) &\equiv wlp(S_1^{C}, wlp(S_2^{C}, R)) \\
&\equiv \text{“By induction hypothesis”} \\
&\quad \neg sp({}^{o}S_1^{C}, \neg wlp(S_2^{C}, R)) \\
&\equiv \text{“By induction hypothesis”} \\
&\quad \neg sp({}^{o}S_1^{C}, sp({}^{o}S_2^{C}, \neg R)) \\
&\equiv \neg sp({}^{o}(S_1\ ;\ S_2)^{C}, \neg R)
\end{aligned}$$

For the local variable declaration the following equivalence holds with a fresh variable w, i.e., $w \notin \mathit{Idf}((\mathbf{var}\ v : T \bullet S_1)^C)$ and $w \in \mathit{Notfree}(R)$.

$$\begin{aligned}
& \mathit{wlp}((\mathbf{var}\ v : T \bullet S_1)^C, R) \\
\equiv\ & (\forall v(\mathit{wlp}(S_1^{v\in T\wedge C[w/v]}, R[w/v])))[v/w] \\
\equiv\ & \text{“By induction hypothesis”} \\
& (\forall v(\neg \mathit{sp}({}^o S_1^{v\in T\wedge C[w/v]}, \neg R[w/v])))[v/w] \\
\equiv\ & \text{“By de Morgan”} \\
& \neg(\exists v(\mathit{sp}({}^o S_1^{v\in T\wedge C[w/v]}, \neg R[w/v])))[v/w] \\
\equiv\ & \neg \mathit{sp}({}^o(\mathbf{var}\ v : T \bullet S_1)^C, \neg R)
\end{aligned}$$

Finally, for nondeterministic choice the following equivalence holds.

$$\begin{aligned}
\mathit{wlp}((\sqcap_{i\in\mathcal{A}} S_i)^C, R) \equiv\ & \wedge_{i\in\mathcal{A}} \mathit{wlp}(S_i^C, R) \\
\equiv\ & \text{“By induction hypothesis”} \\
& \wedge_{i\in\mathcal{A}} \neg \mathit{sp}({}^o {S_i}^C, \neg R) \\
\equiv\ & \text{“Extended de Morgan rule”} \\
& \neg \vee_{i\in\mathcal{A}}\ \mathit{sp}({}^o {S_i}^C, \neg R) \\
\equiv\ & \neg \mathit{sp}({}^o(\sqcap_{i\in\mathcal{A}} S_i)^C, \neg R)
\end{aligned}$$

Assertion (iii): We show this condition by monotonicity of the predicate transformers that form the semantics of the command language constructs. Let R be a predicate without free primed variables.

$$\begin{aligned}
& \mathit{wlp}([\neg \mathit{wlp}(S, \mathit{false})]\ ;\ S, R) \\
\equiv\ & \text{“By definition of sequential composition”} \\
& \mathit{wlp}([\neg \mathit{wlp}(S, \mathit{false})], \mathit{wlp}(S, R)) \\
\equiv\ & \text{“By semantics of specification statement with empty frame”} \\
& \neg \mathit{wlp}(S, \mathit{false}) \Rightarrow \mathit{wlp}(S, R) \\
\equiv\ & \mathit{wlp}(S, \mathit{false}) \vee \mathit{wlp}(S, R) \\
\equiv\ & \text{“By monotonicity of } \lambda R \bullet \mathit{wlp}(S, R) \text{ and the implication } \mathit{false} \Rrightarrow R\text{”} \\
& \mathit{wlp}(S, R)
\end{aligned}$$

Assertion (v): We show this condition by structural induction over the command language. We rely on Lemma A.4 to allow us to work with a specification statement in a trivial context. For a specification statement S of the form $x{:}\,[Q]$ we obtain the following.

$$\begin{aligned}
\mathit{wlp}(S, \mathit{sp}(S, R)) \equiv\ & \forall x'(Q \Rightarrow (\exists x(Q \wedge R))[x/x'][x'/x]) \\
\equiv\ & \forall x'(Q \Rightarrow (\exists x(Q \wedge R))) \\
\Leftarrow\ & \text{“By Lemma A.1, Appendix A”} \\
& R
\end{aligned}$$

For two commands S_1, S_2 the following property of sequential composition

holds.

$$
\begin{aligned}
& wlp((S_1 \,;\, S_2)^C, sp((S_1 \,;\, S_2)^C, R)) \\
\equiv\ & wlp(S_1^C, wlp(S_2^C, sp(S_1^C \,;\, S_2^C, R))) \\
\equiv\ & wlp(S_1^C, wlp(S_2^C, sp(S_2^C, sp(S_1^C, R)))) \\
\Leftarrow\ & \text{“By induction hypothesis and monotonicity of wlp”} \\
& wlp(S_1^C, sp(S_1^C, R)) \\
\Leftarrow\ & \text{“By induction hypothesis”} \\
& R
\end{aligned}
$$

For the local variable declaration the following implication holds.

$$
\begin{aligned}
& wlp((\mathbf{var}\ v : T \bullet S_1)^C, sp((\mathbf{var}\ v : T \bullet S_1)^C, R)) \\
\equiv\ & (\forall v(wlp(S_1^{v \in T \wedge C[w/v]}, (\exists v(sp(S_1^{v \in T \wedge C[w/v]}, R[w/v]))[v/w])[w/v])))[v/w] \\
\equiv\ & (\forall v(wlp(S_1^{v \in T \wedge C[w/v]}, \exists v(sp(S_1^{v \in T \wedge C[w/v]}, R[w/v])))))[v/w] \\
\Leftarrow\ & \text{“By monotonicity and the implication } sp(\cdots) \Rrightarrow \exists v(sp(\cdots))\text{”} \\
& (\forall v(wlp(S_1^{v \in T \wedge C[w/v]}, sp(S_1^{v \in T \wedge C[w/v]}, R[w/v]))))[v/w] \\
\Leftarrow\ & \text{“By induction hypothesis”} \\
& (\forall v(R[w/v]))[v/w] \\
\equiv\ & R
\end{aligned}
$$

For the nondeterministic choice command the following implication holds.

$$
\begin{aligned}
& wlp((\sqcap_{i \in \mathcal{A}} S_i)^C, sp((\sqcap_{i \in \mathcal{A}} S_i)^C, R)) \\
\equiv\ & \wedge_{i \in \mathcal{A}} wlp(S_i^C, \vee_{j \in \mathcal{A}} sp(S_j^C, R)) \\
\Leftarrow\ & \text{“By monotonicity of the ‘wlp’ operator”} \\
& \wedge_{i \in \mathcal{A}}\, wlp(S_i^C, sp(S_i^C, R)) \\
\Leftarrow\ & \text{“By induction hypothesis”} \\
& \wedge_{i \in \mathcal{A}}\, R \\
\Leftarrow\ & R
\end{aligned}
$$

Assertion (iv): From assertion (v) we conclude $wlp(S, sp(S, true)) \equiv true$. With the help of this equivalence we can prove the following chain of predicate equivalences. Let R be a predicate without free primed variables.

$$
\begin{aligned}
wlp(S \,;\, [sp(S, true)], R) \equiv\ & wlp(S, wlp([sp(S, true)], R)) \\
\equiv\ & wlp(S, sp(S, true) \Rightarrow R)) \\
\equiv\ & \text{“By Theorem 6.1, assertion } (v)\text{”} \\
& wlp(S, sp(S, true)) \wedge wlp(S, sp(S, true) \Rightarrow R) \\
\equiv\ & \text{“By Theorem 5.2, assertion } (iii)\text{”} \\
& wlp(S, sp(S, true) \wedge (sp(S, true) \Rightarrow R)) \\
\equiv\ & wlp(S, sp(S, true) \wedge R) \\
\equiv\ & \text{“By Theorem 5.2, assertion } (iii)\text{”} \\
& wlp(S, sp(S, true)) \wedge wlp(S, R)
\end{aligned}
$$

$\equiv$ "By Theorem 6.1, assertion (v)"
$wlp(S, R)$

Assertion (vi): This is a consequence of assertions (i), (ii) and (v).

Assertion (vii): We assume that $S \sqsubseteq_{wlp} S'$. Then the following implications hold for any predicate R without primed variables.

$$\begin{aligned} sp(S', R) &\Rrightarrow \text{"Monotonicity and Theorem 6.1, assertion } (v)\text{"} \\ &\quad sp(S', wlp(S, sp(S, R))) \\ &\Rrightarrow \text{"Monotonicity and } S \sqsubseteq_{wlp} S'\text{ "} \\ &\quad sp(S', wlp(S', sp(S, R))) \\ &\Rrightarrow \text{"Monotonicity and Theorem 6.1, assertion } (vi)\text{"} \\ &\quad sp(S, R) \end{aligned}$$

This means $S \sqsubseteq_{sp} S'$ by Definition 4.3. For the reverse implication note that $S \sqsubseteq_{wlp} S'$ is equivalent to ${}^{o}S \sqsubseteq_{sp} {}^{o}S'$ which is a consequence from Theorem 6.1, assertions (i) and (ii).

Assertion $(viii)$: This is a consequence of the equivalence of $S \sqsubseteq_{wlp} S'$ and ${}^{o}S \sqsubseteq_{sp} {}^{o}S'$, and Theorem 6.1, assertion (vii). □

These equivalences show that weakest precondition and strongest postcondition semantics for commands are equivalent, in other words, they define the same ordering on the command language.

7 Conclusion

Extracting and studying program paths is an important, fundamental concept in static code analysis, program testing, and performance prediction theories and algorithms. Here we have developed a solid foundation for such work by investigating the formal semantics of a language for program paths in depth. For this purpose we devised a minimal command language, described two equivalent command semantics, both of which incorporate a notion of their context, and proved several properties of these semantics including their equivalence in terms of the refinement relation they defined on the command language. We used a partial correctness approach based on weakest liberal preconditions and strongest postconditions in the sense of Dijkstra and Scholten [8,9].

This work provides an essential basis for research into new program analysis techniques. In particular, our own research concerns formal analysis of real-time programs [13,15,16]. In companion papers [18,17,11] we have used the semantics described above as the formal justification for new techniques and algorithms for extracting hard real-time constraints from program code.

Acknowledgement

We wish to thank Robert Colvin for correcting errors in this paper. This research was supported by the Australian Research Council Large Grants A49937045, *Effective Real-Time Program Analysis* and DP209722, *Derivation and timing analysis of concurrent real-time software.*

A Basic Theorems and Lemmas

The following lemmas and theorems were used in the proofs of this paper.

Lemma A.1 *Let R and Q be predicates with $x' \in Notfree(R)$. Then, $R \Rrightarrow \forall x' (Q \Rightarrow \exists x (Q \wedge R))$ holds.*

Proof. The following implication holds for a predicate R and $x' \in Notfree(R)$.

$$\begin{aligned} R &\Rrightarrow (\forall x' \neg Q) \vee R \\ &\equiv \forall x'(Q \Rightarrow R) \\ &\equiv \forall x'(Q \Rightarrow (Q \wedge R)) \\ &\Rrightarrow \forall x'(Q \Rightarrow \exists x(Q \wedge R)) \end{aligned}$$

□

Lemma A.2 *Let x and y be disjoint sets of variables and let Q be a predicate that has free primed variables in x' only. Then, the liberal refinement equivalence $x{:}\,[Q] \sqsupseteq_{wlp} x, y{:}\,[Q \wedge y = y']$ holds.*

Proof. Let R be a predicate without free primed variables. Then the following equivalence holds.

$$\begin{aligned} wlp(x, y{:}\,[Q \wedge y = y'], R) &\equiv \forall x', y'((Q \wedge y = y') \Rightarrow R[x', y'/x, y]) \\ &\equiv \text{“Since } y' \text{ is not free in } Q \text{ and } R\text{”} \\ &\quad \forall x'(Q \Rightarrow R[x'/x]) \\ &\equiv wlp(x{:}\,[Q], R) \end{aligned}$$

□

Lemma A.3 *Let $S \stackrel{def}{=} x{:}\,[Q]$ and $S' \stackrel{def}{=} x{:}\,[W]$ be two commands. Then $S \sqsubseteq_{wlp} S'$ is equivalent to the following implication holding for every predicate R without free primed variables.*

$$\exists x' (W \wedge R[x'/x]) \Rrightarrow \exists x' (Q \wedge R[x'/x])$$

Proof. We first compute $\neg wlp(S, R) \equiv \exists x'(Q \wedge \neg R[x'/x])$. The same can now be done for $wlp(S', R)$ and with the implication $wlp(S, R) \Rrightarrow wlp(S', R)$ we obtain the assertion. □

Lemma A.4 *Let $S \stackrel{def}{=} (\mathbf{var}\, v : T \bullet x, v{:}\,[Q])^C$ with disjoint v and x. Then, S is liberal refinement equivalent to $x{:}\,[C \wedge C[x'/x] \wedge \exists v, v'(v, v' \in T \wedge Q)]$.*

Proof. Let R denote a predicate without free primed variables and let w be a variable identifier such that $w \notin \mathit{Idf}(S)$ and $w \in \mathit{Notfree}(R)$. Then the following equivalence holds.

$$\begin{aligned}
wlp(S,R) &\equiv (\forall v(wlp(x, v{:}\,[Q]^{v\in T\wedge C[w/v]}, R[w/v])))[v/w] \\
&\equiv (\forall v(\forall x', v'((Q \wedge v, v' \in T \wedge C[w/v] \wedge \\
&\qquad\qquad C[w/v][x'/x]) \Rightarrow R[w/v][x'/x])))[v/w] \\
&\equiv (\forall x', v, v'((Q \wedge v, v' \in T \wedge C[w/v] \wedge \\
&\qquad\qquad C[w/v][x'/x]) \Rightarrow R[w/v][x'/x]))[v/w] \\
&\equiv (\forall x'((C[w/v] \wedge C[w/v][x'/x] \wedge \exists v, v'(Q \wedge v, v' \in T)) \Rightarrow \\
&\qquad\qquad R[w/v][x'/x]))[v/w] \\
&\equiv \forall x'((C \wedge C[x'/x] \wedge \exists v, v'(Q \wedge v, v' \in T)) \Rightarrow R[x'/x]) \\
&\equiv wlp(x{:}\,[C \wedge C[x'/x] \wedge \exists v, v'(v, v' \in T \wedge Q)], R)
\end{aligned}$$

□

Lemma A.5 *Let $S \stackrel{def}{=} x{:}\,[Q]$ and $S' \stackrel{def}{=} x{:}\,[W]$ and let v denote a variable with $v \notin \mathit{Idf}(S) \cup \mathit{Idf}(S')$. The sequential composition S ; S' is then liberal refinement equivalent to $x{:}\,[\exists v(Q[v/x'] \wedge W[v/x])]$.*

Proof. Let R be a predicate without free primed variables. Then, the following equivalences hold.

$$\begin{aligned}
& wlp(S\ ;\ S', R) \\
\equiv\ & wlp(S, wlp(S', R)) \\
\equiv\ & \forall x'(Q \Rightarrow (\forall x'(W \Rightarrow R[x'/x]))[x'/x]) \\
\equiv\ & \text{“Provided } v \notin \mathit{Idf}(S) \cup \mathit{Idf}(S') \text{ and } v \in \mathit{Notfree}(R) \text{ ”} \\
& \forall v(Q[v/x'] \Rightarrow (\forall x'(W \Rightarrow R[x'/x]))[v/x]) \\
\equiv\ & \forall v(Q[v/x'] \Rightarrow (\forall x'(W[v/x] \Rightarrow R[x'/x]))) \\
\equiv\ & \forall x', v((Q[v/x'] \wedge W[v/x]) \Rightarrow R[x'/x]) \\
\equiv\ & \forall x'(\exists v(Q[v/x'] \wedge W[v/x]) \Rightarrow R[x'/x]) \\
\equiv\ & wlp(x{:}\,[\exists v(Q[v/x'] \wedge W[v/x])], R)
\end{aligned}$$

□

Lemma A.6 *Assume statements $S_i \stackrel{def}{=} x{:}\,[Q_i]$, $i \in \mathcal{A}$. Then, the nondeterministic choice $(\sqcap_{i\in\mathcal{A}} S_i)$ is liberal refinement equivalent to $x{:}\,[\vee_{i\in\mathcal{A}} Q_i]$.*

Proof. Let R be a predicate without free primed variables. Then the following equivalences hold.

$$\begin{aligned}
wlp(x{:}\,[\vee_{i\in\mathcal{A}} Q_i], R) &\equiv \forall x((\vee_{j\in\mathcal{A}} Q_i) \Rightarrow R[x'/x])) \\
&\equiv \forall x(\wedge_{i\in\mathcal{A}}(Q_i \Rightarrow R[x'/x])) \\
&\equiv \wedge_{i\in\mathcal{A}}(\forall x(Q_i \Rightarrow R[x'/x])) \\
&\equiv wlp((\sqcap_{i\in\mathcal{A}} S_i), R)
\end{aligned}$$

□

References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems (WRTS)*, pages 102–107, 1996.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[3] R. J. R. Back. Refinement calculus, lattices and higher order logic. In M. Broy, editor, *Program Design Calculi*, pages 53–72. Springer-Verlag, 1993.

[4] R.-J. R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[5] B. Carré. Program analysis and verification. In C. T. Sennett, editor, *High-Integrity Software*, chapter 8, pages 176–197. Plenum Press, 1989.

[6] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer Berlin, 1981.

[9] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[10] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174. IEEE Computer Society, 2000.

[11] C. J. Fidge. Timing analysis of assembler code control-flow paths. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods–Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 370–389. Springer-Verlag, 2002.

[12] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[13] S. Grundon, I. J. Hayes, and C. J. Fidge. Timing constraint analysis. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Science Conference*, pages 575–586. Springer-Verlag, 1998.

[14] E. L. Gunter and D. Peled. Path exploration tool. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 405–419. Springer-Verlag, 1999.

[15] I. J. Hayes, C. J. Fidge, and K. Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings—Software*, 148(6):175–186, 2001.

[16] K. Lermer and C. J. Fidge. A formal model of real-time program compilation. *Theoretical Computer Science*, 282(1):151–190, 2002.

[17] K. Lermer, C. J. Fidge, and I. J. Hayes. Extracting execution time constraints from real-time programs. Technical Report 02-30, Software Verification Reseach Centre, The University of Queensland, Oct 2002.

[18] K. Lermer, C. J. Fidge, and I. J. Hayes. A theory for execution time derivation in real-time programs. Technical Report 02-13, Software Verification Reseach Centre, The University of Queensland, April 2002.

[19] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[20] C. Morgan and T. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.

[21] C. Morgan and T. Vickers. *On the Refinement Calculus*. Springer-Verlag, 1994.

[22] S. L. Pfleeger. *Software Engineering: The Production of Quality Software*. Macmillan, second edition, 1991.

[23] J. von Wright. Program inversion in the refinement calculus. *Information Processing Letters*, 37(2):95–100, 1991.

[24] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31:105–135, 1994.